Table of Contents

1 C Programming for Embedded Systems
- 1.1 Structure Types
- 1.2 Enumerated Types
- 1.3 Bit Manipulation Operators
- 1.4 Function Pointers
- 1.5 Function Pointers and Callback Functions
- 1.6 Volatile Reserved Word
- 1.7 __attribute__
- 1.8 C and Const Correctness

1 C Programming for Embedded Systems

In this section we will review some features of the C programming language that are useful in embedded systems program, and will introduce some new features that you may not have been exposed to before.

1.1 Structure Types[1]

You should have learned all about structure types in CSE220. Take the time now to review Struct.c[2] on the course website.

1.2 Enumerated Types[3]

Suppose we wish to "create" our own data type to represent colors. The traditional way of doing this in C is to use `#define` preprocessor macros and `typedef`,

```
#define COLOR_RED   0
#define COLOR_GREEN 1
#define COLOR_BLUE  2
typedef unsigned int color;
color c = COLOR_RED;
color d = c;
c = 99; // Error. What color is c?
```

With the addition of the `const` reserved word to ANSI C the use of `#define` is discouraged[4], so a better way to do this would be,

```
int const[5] COLOR_RED   = 0;
int const COLOR_GREEN = 1;
int const COLOR_BLUE  = 2;
typedef unsigned int color;
color c = COLOR_RED;
color d = c;
c = 99; // Error. What color is c?
```

There's nothing necessarily wrong with this, but enumerated types provide a better way with a few minor advantages—one of which depends on your IDE. Consider the declaration of an enumerated type for colors,

1 Good Tutorial: http://www.programiz.com/c-programming/c-structures.
2 See Struct.c posted on the course website.
3 Good Tutorial: http://www.programiz.com/c-programming/c-enumeration. See Enum.c posted on the course website.
4 Why? Because `const` enables the C compiler to perform more accurate static type-checking.
5 `const int COLOR_RED = 0;` is equivalent to `int const COLOR_RED = 0;`

Advantages to `enum`s,

1. Proper use of enumerated types should improve the readability of the code (when you create an enumerated type in C, you are not really creating a "new type", but rather, an enumerated type is just a mapping from magic numbers to identifiers).
2. `enum` values can be auto-generated; by default, the values are 0, 1, 2, ..., so this saves a bit of typing.
3. Some compilers can emit a warning if a `switch` statement does not include `case`s for all enumerated values.
4. In some IDE's, the editor will syntax color enumerated values. Also, the debugger will display the identifier for the enumerated values rather than the integer values. This last advantage is probably the most useful one.

1.3 Bit Manipulation Operators[6, 7]

There are six bit manipulation operators in C.

1.3.1 Bitwise AND Operator: &

Remember: `&&` is logical AND.

```
uint8_t[8] x = 0x3C, y = 0xA5, z;    0011 1100 = 0x3C (x)               0010 0100 = 0x24 (z)
z = x & y;                         & 1010 0101 = 0xA5 (y)             & 0011 1100 = 0x3C (x)
z &= x;                          ==> 0010 0100 = 0x24 (z = x & y)  ==>  0010 0100 = 0x24 (z &= x)
```

1.3.2 Bitwise OR Operator: |

Remember: `||` is logical OR.

```
uint8_t x = 0x3C, y = 0xA5, z;     0011 1100 = 0x3C (x)               1011 1101 = 0xBD (z)
z = x | y;                       | 1010 0101 = 0xA5 (y)             | 0011 1100 = 0x3C (x)
z |= x;                        ==> 1011 1101 = 0xBD (z = x | y)  ==> 1011 1101 = 0xBD (z |= x)
```

1.3.3 Bitwise Complement Operator: ~

Forms the one's complement of the operand.

```
uint8_t x = 0x3C; z = ~x;       ~0011 1100 ==> 1100 0011 = 0xC3
```

1.3.4 Bitwise Exclusive-OR (XOR) Operator: ^

`|` is called the bitwise inclusive-OR operator. Exclusive-OR is 0 if the operands are the same and 1 if the operands are different.

```
uint8_t x = 0x3C, y = 0xA5, z;   0011 1100 = 0x3C (x)             1001 1001 = 0x99 (z)
z = x ^ y;                     ^ 1010 0101 = 0xA5 (y)           ^ 0011 1100 = 0x3C (x)
z ^= x;                          1001 1001 = 0x99 (z = x ^ y)  ==> 1010 0101 = 0xA5 (z ^= x)
```

1.3.5 Bitwise Shift Left Operator: <<

The operation `a << n` will shift the bits in variable a to the left n times filling new bit positions with 0.

```
uint8_t x = 0x3C, z;
z = x << 3;                  0011 1100 << 3 ==> 1110 0000 = 0xE0
z <<= 2;                     1110 0000 << 2 ==> 1000 0000 = 0x80
```

On unsigned integer types consisting of b-bits, shifting a left by n will multiply a by 2^n, but you must be careful to avoid the overflow which will occur when $a << n$ is too large to be represented in b-bits. Overflow can be avoid by ensuring that $a \cdot 2^n \le 2^b - 1$ or $a \le (2^b - 1)/2^n$.

Example, a is a `uint16_t` variable and we wish to multiply $a = 14{,}567$ by $n = 16 = 2^4$ using the left-shift trick. Will this result in overflow? To avoid overflow, it must be true that $a \le (2^b - 1)/2^n$ or $14{,}567 \le (2^{16} - 1)/2^4$ or $14{,}567 \le 4095.9375$, which is false; therefore, overflow will occur.

6 http://www.cprogramming.com/tutorial/bitwise_operators.html
7 http://en.wikipedia.org/wiki/Bitwise_operations_in_C
8 `typedef unsigned char uint8_t`; See "global.h" in *newbbproj.zip*.

On b-bit signed integer types, shifting a nonnegative integer left by n will also multiply by 2^n, but you must be careful not to trash the sign bit. The largest nonnegative integer that may be safely shifted left n times is $(2^{b-1} - 1)/2^n$. It is never wise to try this trick on a negative integer (the result is undefined).

1.3.6 Bitwise Shift Right Operator: >>

The operation `a >> n` will shift the bits in variable a to the right n times, filling vacated bit positions with 0.

```
uint8_t x = 0x3C, z;
z = x >> 2;                  0011 1100 >> 2 ==> 0000 1111 = 0x0F  (which is int(60 / 4))
z >>= 3;                     0000 1111 >> 3 ==> 0000 0001 = 0x01  (which is int(15 / 4))
```

On unsigned integer types, shifting right by n will divide (integer division) the operand by 2^n. On signed integer types, shifting a nonnegative integer right by n will also divide by 2^n. It is never wise to try this trick on a negative integer (the result is undefined[9]).

1.3.7 Setting Bits

To "set" a bit means to put a 1 into a specific bit position. Boolean algebra: $a \vee 0 = a$, $a \vee 1 = 1$.

Example: Set bit 3 of `uint8_t` variable x (mask must have 1 in bit 3 and 0's in other bits; use bitwise OR).

```
x |= 1 << 3;    or    x |= 0x08;
```

Example: Set bits 3 and 6 of x (mask must have 1's in bits 3 and 6 and 0's in other bits).

Example: Set bit n of x (mask must have 1 in bit n and 0's in other bits).

Example: Set bits n and m of x (mask must have 1's in bits n and m and 0's in other bits).

1.3.8 Clearing Bits

To "clear" a bit means to put a 0 into a bit position. Boolean algebra: $a \wedge 0 = 0$, $a \wedge 1 = a$.

Example: Clear bit 3 of `uint8_t` variable x (mask must have 0 in bit 3 and 1's in other bits; use bitwise AND).

```
x &= ~(1 << 3);    or    x &= 0xF7;
```

Example: Clear bits 3 and 6 of x (mask must have 0 in bits 3 and 6 and 1's in other bits).

Example: Clear bit n of x (mask must have 0 in bit n and 1's in other bits).

Example: Clear bits n and m of x (mask must have 0's in bits n and m and 1's in other bits).

---

9 Most ISA's contain at least two shift right instructions: one to perform a logical shift right and one to perform an arithmetic shift right. Logical shift right fills vacated bit positions with 0's and most certainly would not work on a negative integer. Arithmetic shift right fills vacated bit positions with the sign bit and would therefore work on a negative integer. However, which of these is used is compiler-dependent, so unfortunately, you should not depend on it working if you wish to write portable C code.

1.3.9 Testing Bits for Set

Example: Test bit n of x to see if it is set (mask must have 1 in bit n and 0's in other bits; use bitwise AND).

```
if (x & (1 << n) != 0) { ... }    or    if (x & (1 << n)) { ... }
```

1.3.10 Testing Bits for Clear

Example: Test bit n of x to see if it is clear (mask must have 1 in bit n and 0's in other bits; use bitwise AND).

1.4 Function Pointers[10]

The name of a C function is an *identifier*. Every function has an associated *memory address* which is the address of the first instruction of the function. A *pointer variable* is a variable that contains the *memory address* of an identifier. Putting all of this together means that we can create a pointer variable which contains the address of a function, i.e., a *function pointer*. The basic syntax for declaring a function pointer variable is (nasty),

```
funct-return-type (*funct-ptr-var-name)(parameter-list)
```

For example,

```
// foo() is a function has an int and double parameter and returns an int
int foo(int x, double y) { ... }

// bar() is a function that has an int and int* parameter and returns a pointer to an int
int *bar(int x, int *y) { ... }

// baz() is a function that has an int and int* parameter and returns a pointer to an int
int *baz(int x, int *y) { ... }

// Define a new type named fptr_i_ip_ip t which is a function pointer to a function that has parameters of
// types int and int * and the function returns an int *. The i part of fptr_i_ip_ip_t specifies that the
// first parameter is an int; the first ip part specifies that the second parameter is an int pointer;
// the third ip specifies that the return type is an int pointer. What is the data type of a function pointer
// variable defined to be of type fptr_i_ip_ip_t? It is: int * (*)(int, int*).
typedef int * (*fptr_i_ip_ip_t)(int, int *);

// I prefix my parameter names in c with a leading p_. gromulate() has two parameter: the first, named
// p_some_funct, is a function pointer to a function that has int and int * parameters and return a pointer
// to an int; the second parameter, p_a, is an int.
int *gromulate(fptr_i_ip_ip_t p_some_funct, int p_a)
{
    int x = 1;

    // Calls the function that some_funct points to. Passes a (an int) and the address of x (an int *).
    // some_funct() returns a pointer to an integer, so we return that pointer.
    return p_some_funct(p_a, &x);
}

void glorp()
{
    // Defines funct_ptr to be a function pointer variable to a function which has parameters of types int and
    // double and returns an int. And, what is the data type of funct_ptr? It is int (*)(int, double).
    int (*funct_ptr)(int, double);

    // Make funct_ptr point to foo().
    funct_ptr = foo;

    // Call foo() indirectly via funct_ptr. Passes an int and a double. Assigns the returned int to a.
    int a = funct_ptr(2, 3.1);

    // Call gromulate() passing a pointer to bar() as the argument. This indirectly calls bar(). Assigns the
    // integer pointer returned from bar() to p.
    int *p = gromulate(bar, 9);
```

10 http://www.cprogramming.com/tutorial/function-pointers.html

```
    // Call gromulate() passing a pointer to baz() as the argument. This indirectly calls baz(). Assigns the
    // integer pointer returned from baz() to q.
    int *q = gromulate(baz, -5);
}
```

1.5 Function Pointers and Callback Functions

A *callback*[11] *function* (or simply *callback*) is a function f passed an argument to some other function g that is to be called by g when some event occurs that the f needs to be notified of. Callbacks can be blocking (synchronous) or nonblocking (asynchronous). A blocking callback g does not return until f is called (back). A nonblocking callback would one where g returns before calling f, although g will call f at some future time. We will only discuss nonblocking callbacks.

A very common use of a callback—and this is how we will use them—is for a application (user) function f to request the operating system or a library function g to call f when some event occurs that is detected by the operating system or library function. Here is a typical example,

```
//*****************************************************************************************************************
// FILE: os_frobble.h
//
// DESCRIPTION
// os_frobble.h contains a typedef for a callback function pointer type.
//*****************************************************************************************************************
#ifndef OS_FROBBLE_H
#define OS_FROBBLE_H

// callback_i_v_t is a new type[12] which is a function pointer to a function with an int parameter that returns
// nothing.
typedef void (*callback_i_v_t)(int);

#endif
```

```
//*****************************************************************************************************************
// FILE: os_frobble.c
//
// DESCRPITION
// os_frobble.c contains some OS functions.
//*****************************************************************************************************************
#include "os_frobble.h"

// g_callback is a static global pointer to a void function with one int parameter. g_callback currently points
// to nothing as it is initialized to null.
static callback_i_v_t g_callback = 0;

// This function saves p_callback in g_callback so that when some event occurs, os_frobble_event_handler()
// will call p_callback.
void os_frobble_callback_request(callback_i_v_t p_callback)
{
    g_callback = p_callback;
}

// This function gets called when some external event happens that the OS detects. It will call the user callback
// function stored in g_callback to let it know that the event happened.
void os_frobble_event_handler()
{
    int x = assume x is somehow assigned a value;

    // Check that g_callback points to something before calling the callback function. This will avoid
    // attempting to access a null pointer (which is generally, a bad thing).
    if (g_callback) g_callback(x);
}
```

11 https://en.wikipedia.org/wiki/Callback_(computer_programming)

12 It's really not a new type, but it is okay to think of it as being one.

```
//*****************************************************************************************************************
// FILE: gromulate.c
//
// DESCRIPTION
// gromulate.c contains user (application code).
//*****************************************************************************************************************
#include <os_frobble.h>

// This function initializes the gromulate module and in the process, it requests the os_frobble module to call
// gromulate_callback() when some event occurs.
void gromulate_init()
{
    os_frobble_callback_request(gromulate_callback);
    ...
}
// This callback function will be called by the os_frobble module when the external event happens.
void gromulate_callback(int p)
{
    // Do something to handle the event...
}
```

Essentially, the *gromulate* user module is delegating the responsibility of detecting the external event to the *os_frobble* operating system module, which notifies the *gromulate* module—via the callback function—when the event has occurred. We will see examples of this, and use this technique, when we talk about interrupts later in the course.

1.6 Volatile Reserved Word[13]

The `volatile` reserved word was added to the C language standard specifically to support real-time and embedded systems programming. Why? Suppose we are accessing an 8-bit hardware register named SETTA that controls a general-purpose I/O pin. We want to write a 1 to bits 0 and 3 of this register (without affecting the other bits) and later on we are going to read the value of bit 3. Assume the address of SETTA is 0x4000_04C0.

```
[001]   #define SETTA (*(uint8_t *))(0x400004C0); // SETTA is equiv to the dereferenced value of addr 0x4000_04C0.
[002]   int x;                                   // x is an uninitialized variable
[003]   SETTA |= 0x09;                           // Set bits 0 and 3
...
[025]   x = (SETTA & 0x08) >> 3;                 // Set x to bit 3 of SETTA.
```

This would be the C code after the preprocessor expands the SETTA macro,

```
[001] int x;
[002] (*(unsigned char *))(0x400004C0) |= 0x09;
...
[023] x = ((*(unsigned char *))(0x400004C0) & 0x08) >> 3;
```

Where the second line is equivalent to,

```
(*(unsigned char *))(0x400004C0) = (*(unsigned char *))(0x400004C0) | 0x01;
```

Now, consider the assembly language code that a C compiler might generate for these statements (this is pseudo-assembly language—ah, I just love assembly language, such power),

```
; Line 1: int x;
sub      sp, sp, 4          ; Subtract 4 from sp to allocate 4 bytes on stack for local variable x
; Line 2: SETTA |= 0x01;
laddr    r0, 0x400004C0     ; r0 ← &SETTA
load     r1, [r0]           ; r1 ← SETTA
ori      r1, r1, 9          ; r1 ← SETTA | 0x09
store    [r0], r1           ; SETTA ← SETTA | 0x09
```

13 http://publications.gbdirect.co.uk/c_book/chapter8/const_and_volatile.html

Suppose an external hw event clears bit 3 of SETTA after the line above is executed

```
; Line 23: x = (SETTA & 0x08) >> 3;
andi     r1, r1, 8         ; r1 ← SETTA & 0x08
lsr      r1, r1, 3         ; r1 ← (SETTA & 0x08) >> 3
store    [r0], r1          ; SETTA ← (SETTA & 0x08) >> 3
```

Now trace through this code,

So, the problem is that the compiler—in optimizing the code[14]—had no idea that SETTA could change value between lines 2 and 23, and for normal microcontroller registers that would not happen, but SETTA is not just any other microcontroller register. It is connected to a peripheral which can change the value of any bit in SETTA at any time. So, between the `ori` and `andi` instructions some event could occur that would change the value of bit 3 from 0 to 1 or 1 to 0. Therefore, the value assigned to x would be the *old* value of bit 3 of SETTA. This could, of course, lead to a bug, which we would like to prevent. Thus the addition of the `volatile` reserved word to the language.

What `volatile` does is inform the compiler that the object to which `volatile` is applied is subject to change in ways that the compiler cannot know. This forces the compiler to always access the memory location to load the value of the object every time it is used. So to define register SETTA we would write,

```
#define SETTA (*(volatile uint8 *))(0x400004C0);
```

The generated assembly language code would be,

```
; Line 1: int x;
sub      sp, sp, 4         ; Subtract 4 from sp to allocate 4 bytes on stack for local variable x
; Line 2: SETTA |= 0x01;
laddr    r0, 0x400004C0    ; r0 ← &SETTA
load     r1, [r0]          ; r1 ← SETTA
ori      r1, r1, 9         ; r1 ← SETTA | 0x09
store    [r0], r1          ; SETTA ← SETTA | 0x09
```

Suppose an external hw event clears bit 3 of SETTA after the line above is executed

```
; Line 23: x = (SETTA & 0x08) >> 3;
load     r1, [r0]          ; r1 ← SETTA
andi     r1, r1, 8         ; r1 ← SETTA & 0x08
lsr      r1, r1, 3         ; r1 ← (SETTA & 0x08) >> 3
store    [r0], r1          ; SETTA ← (SETTA & 0x08) >> 3
```

14 Of course, if we turned off optimization, then this problem would not occur, but in general, optimization is a good thing because it will make the generated machine language code more efficient.

Another, more problematic situation, would occur if we wrote something like this (where SETTA is not `volatile`),

```
// Loop while bit 3 of SETTA is 0.
while (~SETTA & 0x08) {
    ...
}
...
```

The optimized assembly language code for this loop is (note, we only r1 with ~SETTA & 0x08 once),

```
laddr    r0, 0x400004C0    ; r0 ← &SETTA
load     r1, [r0]          ; r1 ← SETTA
not      r1, r1            ; r1 ← ~SETTA
andi     r1, 8             ; r1 ← ~SETTA & 0x08
@beginloop:
cmp      r1, 0             ; Drop out of loop
beq      @endloop          ;   if ~SETTA & 0x08 is zero
...                        ; During the 12th pass of this loop, an external event changes bit 3 of SETTA to 1.
bra      @beginloop        ; Continue looping.
@endloop:
...
```

With the `volatile` reserved word applied to SETTA the assembly language code would be (note, we load r1 with ~SETTA & 0x08 each time the loop condition is evaluated),

```
@beginloop:
laddr    r0, 0x400004C0    ; r0 ← &SETTA
load     r1, [r0]          ; r1 ← SETTA
not      r1, r1            ; r1 ← ~SETTA
andi     r1, 8             ; r1 ← ~SETTA & 0x08
cmp      r1, 0             ; Drop out of loop
beq      @endloop          ;   if ~SETTA & 0x08 is zero
...                        ; During the 12th pass of this loop, an external event changes bit 3 of SETTA to 1.
bra      @beginloop        ; Continue looping.
@endloop:
...
```

Voilà! Bugs be gone. When we talk about interrupt service routines I will explain another situation where using `volatile` is critical.

1.7 __attribute__

[Ref: CW-BUILD, §5.4] Most C compilers have nonstandard ways of specifying additional information to the compiler, which it uses when compiling the code. For example, Microsoft C and C++ compilers introduced the `__declspec`[15] extension to the C language; it not part of the language standard but many other C and C++ compilers support it.

The GNU C compiler uses the `__attribute__`[16] extension to do this. CodeWarrior uses the GNU C compiler on the backend to compile your code, so CW supports some, but not all, of the GCC `__attribute__` extensions. The general syntax of an __attribute__ specification is,

`__attribute__((`*attribute-list*`))`

where the *attribute-list* is a possibly empty, comma-separated sequence of *attributes*. We will only discuss the attributes that may be applicable to our CSE325 projects.

`alias`

Causes the declaration to be emitted to the linker as a new name for another symbol, e.g.,

15 https://msdn.microsoft.com/en-us/library/dabb5z75.aspx
16 https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html#Attribute-Syntax

```
// void __f() { ... }

...

__attribute__((alias("__f"))) void f();

int main()
{
    f();  // Calls __f()
    return 0;
}
```

On the surface, this may not seem to useful. However, consider the use of `alias` and `weak` in *bare_startup.c*,

```
void bare_default_isr()
{
    __asm("bkpt");
}

// Weak definitions of interrupt service routines point to bare_default_isr if not implemented.
void nmi_isr()       __attribute__((weak, alias("bare_default_isr")));
void hardfault_isr() __attribute__((weak, alias("bare_default_isr")));
void svc_isr()       __attribute__((weak, alias("bare_default_isr")));
...
void portscd_isr()   __attribute__((weak, alias("bare_default_isr")));

// Define and initialize the interrupt vector table. Place the vector table in its own section named .vectortable
// in the .o file. The linker script will use this section name to make sure the vector table is placed in the
// flash at 0x0000_0000.
void (* const bare_vector_table[])() __attribute__((section(".vectortable"))) = {
    (void *)&__init_sp,   // Initial value of the SP register
    bare_startup,         // Reset vector will jump here and begin executing code
    nmi_isr,              // Nonmaskable interrupt service routine
    hardfault_isr,        // Hardfault interrupt service routine
    ...
    portscd_isr           // Single interrupt vector for Port C and Port D interrupt
};
```

The function declaration employing the `weak` and `alias` attributes, e.g., *nmi_isr*(), declares a function that provides a default implementation, which can be overwritten by the user. That is, if the user defines *nmi_isr*(), that function definition will be called in place of the *nmi_isr*() function that is declared `weak`. The `alias` attribute specifies that *nmi_isr*() is an alias for *bare_default_isr*(), so when *nmi_isr*() is called, *bare_default_isr*() will be the function that is *actually* called. Putting `weak` and `alias` together, then, specifies that if *nmi_isr*() is not overwritten, *bare_default_isr*() will be called; if *nmi_isr*() is overwritten, then the user can provide their own code that will be executed when *nmi_isr*() is called.

`interrupt`
Specifies that the function is an interrupt service routine (ISR) so the compiler will generate special prologue and epilogue code appropriate for ISR's. I'll explain this in more detail when we discuss interrupts. For example,

```
__attribute__((interrupt)) timer0_isr() { ... }
```

`naked`
This is an ARM-specific attribute which permits the compiler to construct the requisite function definition, while allowing the body of the function to be assembly code. The function will not have the usual prologue/epilogue code that saves registers on the stack at the beginning of the function, restores registers at the end, and returns from the function. For example,

```
void bar()
{
        __asm("bl bar");
}
```

```
__attribute__((naked)) void foo()
{
        __asm("bl foo");
}
```

Consider, the C compiler-generated assembly language,

```
bar:
    push     {r7,lr}
    add      r7, sp, #0
    bl       bar
    mov      sp, r7
    pop      {r7,pc}

foo:
    bl       foo
```

`section`
Normally, for a function named *foo*(), the GNU C compiler will place the instructions for *foo*() in the *.text* section of the *.o* file. Our CW project settings are configured such that each function is placed in its own unique text section, e.g., *foo*() will be output to *.text.foo*. There are times when it is desirable to specify the name of the section in which the object code is placed. For example, in the code for *bare_startup.c*, scroll to line 98 and you will see,

```
void (* const bare_vector_table[])() __attribute__((section(".vectortable"))) = {
```

which specifies to the compiler that the contents of *bare_vector_table*[] are to be written to a section named *.vectortable* in *bare_startup.o.* This section is used in the *MKL46Z256_flash.ld* linker script to ensure that *bare_vector_table*[] is written to the *m_interrupts* text segment.

`unused`
Specifies that the programmer is aware that a variable or function parameter is not used within the body of a function, e.g.,

```
int some_funct(int x, __attribute__((unused)) double y)
{
    return x + 99;
}
```

Note, this attribute would not be required if we had not enabled the **Extra warnings (-Wextra)** check box in Step 11 of §3.5 of Project 1. You may wonder why one would go to the trouble of defining a function parameter that is not used in the function. I find this attribute to be useful during debugging and testing at times when I want to see if I can get my code to run correctly when I do not pass an argument for *y*; the attribute allows me to try this out without having to locate and modify all calls to *some_funct*().

`weak`[17]
When applied to a function, `weak` causes the function to be emitted to the linker as a "weak" symbol rather than as a "strong" symbol. This is primarily useful in defining library functions containing default code, that may be overwritten by the user. For example,

```
// lib.hpp
void lib_funct();
```

```
// lib.cpp
#include "lib.hpp"
__attribute__((weak)) void lib_funct()
{
    g_delay--;  // Decrement the value of a global variable
}
```

17 https://en.wikipedia.org/wiki/Weak_symbol

```
// user.hpp
void lib_funct();
```

```
// user.cpp
#include "user.hpp"
void lib_funct()
{
    g_delay++;  // Increment the value of a global variable
}
```

```
#include "user.hpp"
int main()
{
   lib_funct(); // Calls lib_funct() in user.cpp which overwrites default lib_funct() in lib.cpp.
}
```

Now, if *lib_funct*() is not overridden in *user.cpp* (delete *user.hpp* and *user.cpp*),

```
// lib.hpp
void lib_funct();
```

```
// lib.cpp
#include "lib.hpp"
__attribute__((weak))__ void lib_funct()
{
    g_delay--;  // Decrement the value of a global variable
}
```

```
int main()
{
   lib_funct(); // Calls lib_funct() in lib.cpp because it is not overwritten
}
```

1.8 C and Const Correctness

The reserved word `const` is a type qualifier and when applied to a variable tells the compiler variable is read-only, i.e., cannot be changed,

```
int w;             // w is an int variable and is uninitialized. It's value is undefined.
int x = 10;        // x is an int variable and is initialized to 10.
int const y;       // Error. const variables must be initialized when defined.
int const z = 10;  // z is a read-only int variable that cannot be changed. It's data type is "int const".
w = 20;            // Okay. w is now 20.
x = 20;            // Okay. x is now 20.
y = 20;            // Error. const variable y is read-only.
z = 20;            // Error. const variable z is read-only.
```

A function's formal parameters are variables and can also be defined as `const`,

```
void foo(int x, int const y)
{
    x = 5;   // Okay. x is now 5.
    y = 15;  // Error. const variable y is read-only.
}
```

```
void bar()
{
    int a = 10;  // a is an int variable with value 10.
    int b = 20;  // b is an int variable with value 20.
    foo(a, b);   // The value of a (10) is copied into x and the value of b (20) is copied into y.
}
```

Since C has only one parameter passing technique (pass by-value), when *foo*() is called, the values of a and b are copied into the memory locations allocated for x and y. Within *foo*(), the value of x can be changed but the value of y cannot.

Note that changing x does not affect the value of a, i.e., on return from *foo*(), the value of a is still 10. The keyword `const` may also be applied to pointers,

```
int main()
{
    // x and y are non-const ints.
    int x = 1, y = 2;

    // C_INT1 and C_INT2 are const ints, i.e., they are read-only.
    int const C_INT1 = 3, int const C_INT2 = 4;

    // Okay. The data type of ptr1 is pointer to int, and it points to an int (x).
    int *ptr1 = &x;

    // Okay. Changes x to 5 (indirection via ptr1).
    *ptr1 = 5;

    // Warning. The data type of ptr2 is pointer to int, but it points to a const int.
    int *ptr2 = &C_INT1;

    // Questionable. Behavior is undefined. On my Linux system, the program crashes with a segmentation fault.
    *ptr2 = 5;

    // Okay. ptr_const1 is an initialized non-const pointer to a const int (C_INT1).
    int const *ptr_const1 = &C_INT1;

    // Error. The const int to which ptr_const1 points is read-only.
    *ptr_const1 = 5;

    // Okay. Since ptr_const1 is a non-const pointer it can be changed to point to a different const int.
    ptr_const1 = &C_INT2;

    // Okay. ptr_const2 is an uninitialized non-const pointer to a const int.
    int const *ptr_const2;

    // Okay. ptr_const2 is now an initialized non-const pointer to a const int (C_INT2).
    ptr_const2 = &C_INT2;

    // Error. The const int to which ptr_const2 points is read-only.
    *ptr_const2 = 5;

    // Okay. Since ptr_const2 is a non-const pointer it can be changed to point to a different const int.
    ptr_const2 = &C_INT2;

    // Okay. ptr_const3 is an uninitialized non-const pointer to an int (x). This seems like it might be illegal
    // since the data type of the thing to which ptr_const3 points is supposed to be "int const" but what happens
    // is that as far as ptr_const3 is concerned, x is now treated as const (read-only).
    int const *ptr_const3 = &x;

    // Error. Since x, via ptr_const3, is read-only, the value of x cannot be changed by dereferencing ptr_const3.
    *ptr_const3 = 5;

    // Okay. const_ptr1 is an initialized const pointer to an int (x).
    int * const const_ptr1 = &x;

    // Error. const pointer const_ptr1 is read-only and cannot be changed to point to something else.
    const_ptr1 = &y;

    // Okay. The data type of the thing to which const_ptr1 points (x) is int. x is now 5.
    *const_ptr1 = 5;

    // Okay. const_ptr2 is a const pointer to an int. const pointers do not have to be initialized when defined
    // and if not, are initialized to NULL. Note that const_ptr2 cannot be changed (see the next statement).
    int * const const_ptr2;

    // Error. const pointer const_ptr2 is read-only and cannot be changed to point to something else.
    const_ptr2 = &x;

    // Okay. const_ptr_const1 is a const pointer to a const int. const pointers do not have to be initialized
    // when defined and if not, are initialized to NULL. Note that const_ptr_const1 cannot be changed (see the
    // next statement).
    int const * const const_ptr_const1;
```

```
    // Errors. Since const_ptr_const1 is a read-only const pointer, it cannot be changed to point to something
    // else.
    const_ptr_const1 = &x;
    const_ptr_const1 = &C_INT1;

    // Okay. const_ptr_const2 is a const pointer to a const int. Even though x is not an const int, as far as
    // const_ptr_const2 is concerned, the int to which it points (x) is now const and cannot be changed.
    int const * const const_ptr_const2 = &x;

    // Error. The const int to which const_ptr_const2 points (x) is read-only and cannot be changed.
    *const_ptr_const2 = 10;

    // Okay. const_ptr_const3 is a const pointer to a const int (C_INT1).
    int const * const const_ptr_const3 = &C_INT1;

    // Error. The const int to which const_ptr_const3 points (C_INT1) is read-only and cannot be changed.
    *const_ptr_const3 = 10;

    // Error. Since const_ptr_const3 is a const pointer, it cannot be changed to point to something else.
    const_ptr_const3 = &C_INT2;
    return 0;
}
```

Note that making a pointer to a `const` point to a non-`const` is legal, but the non-`const` is treated as if it were `const`, i.e., it is read-only. It is also legal to make a pointer to a non-`const` point to a `const`, but the behavior is undefined[18]. For example, gcc will complain with a warning message about non-`const` pointer *ptr2* being made to point to a `const` C_INT1, but if you ignore the warning and proceed to attempt to change the value of C_INT1 by dereferencing *ptr2*, then be prepared for something bad to happen—in this case, my program crashed with a segmentation fault.

The elements of an array may be `const`,

```
typedef int rgb_t[3];  // A color expressed as Red, Green, and Blue components.

int main()
{
    rgb_t const red;                    // Error. red is const and must be initialized at definition,
    rgb_t const green = { 0, 255, 0 };  // Okay. green is initialized and its data type is actually "int const *".
    green[0] = 255;                     // Error. You cannot turn green into yellow, since green[0] is const.
    return 0;
}
```

The primary use of `const` is to practice defensive programming, i.e., to reduce your chances of introducing bugs into your code. If something is not going to be changed or should not be changed, make it `const`. When to use `const`? Whenever you can!

Finally, in the old days, numeric constants were traditionally created using the `#define` preprocessor directive, but today, the recommendation is to use `const` for a variety of reasons, one of the best of which is that the name associated with a `#define` disappears after the preprocessor finishes its job, so consequently, when debugging you will essentially be dealing with magic numbers. With `const` and exporting symbolic debugging information to your executable, you will see the names of your `const`s rather than their values. That alone is good enough reason to use `const`.

18 Which means it is implementation-specific; you should not count on your code behaving in the same way when you change from the Microsoft Visual Studio compiler to the GCC C compiler

References

[ARM-ARM] *ARM Architecture Reference Manual*, Rev I., ARM, 2005.

[ARM-CALL] *Procedure Call Standard for the ARM Architecture*, Rev. E 2.09, ARM, 2012.

[ARM-ISQR] *ARM and Thumb-2 Instruction Set Quick Reference Card*, Rev. M, ARM, 2008.

[ARM-M0+DUG] *Cortex-M0+ Device User's Guide*, ARM, 2012.

[ARM-M0+TRM] *Cortex-M0+ Technical Reference Manual*, Rev. r0p1, ARM, 2012.

[ARM-V6M] *ARMv6-M Architecture Reference Manual*, Rev. C, 2010.

[BARR] Barr, M., *Programming Embedded Systems in C and C++*, O'Reilly, 1999.

[BERGER] Berger, A, *Embedded Systems Design*, CMP Books, 2002.

[CW-BUILD] *CodeWarrior Kinetis Build Tools Reference Manual*, Rev. 10.6, Freescale, 2014.

[FRDM-KL46-PIN] *FRDM-KL46Z Pinout.*

[FRDM-KL46Z-QSG] *FRDM-KL46Z Quick Start Guide*, Rev. 1, Freescale, 2013.

[FRDM-KL46-SCH] *FRDM-KL46Z Schematic*, Rev. C, Freescale, 2013.

[FRDM-KL46Z-UM] *FRDM-KL46Z User's Manual*, Rev. 1.0, Freescale.

[HEATH] Heath, S., *Embedded Systems Design*, 2nd ed, Newnes, 2003.

[KL46-SDS] *Kinetis KL46 Sub-Family Data Sheet*, Rev. 5, Freescale, 2014.

[KL46-SRM] *Kinetis KL46 Sub-Family Reference Manual*, Rev. 3, Freescale, 2013.

[NOER] Noergaard, T., *Embedded Systems Architecture, 1st ed*, Newnes, 2005.

[OPENSDA] *OpenSDA User's Guide*, Rev. 0.93, Freescale, 2012.

[VAHID] Vahid, F., and Givargis, T., *Embedded System Design*, John Wiley & Sons, 2002.

[WHITE] White, E., *Making Embedded Systems*, O'Reilly, 2011.